Virgilian Poetry Generation with LSTM Networks

Stanford CS224N Custom Project

August Burton
Department of Bioengineering
Stanford University
aburton6@stanford.edu

Jonathan Merchan
Departments of Computer Science and Classics
Stanford University
jmerchan@stanford.edu

Abstract

Virgil's poetry has an enduring legacy both as a cultural classic and teaching tool. Furthermore, his works are a gold standard for dactylic hexameter, a complex meter used in classical poetry. Advances in natural language processing provide new avenues for engaging with Virgil's poems. In 2019, Lamar and Chambers claimed that an encoder-decoder long short-term memory (LSTM) network could generate Greek poetry in the style of Homer with reasonably accurate meter but poor semantics after being trained exclusively on Homer. In this paper, we aimed to replicate and, if possible, improve on Lamar and Chamber's work by developing a model that generates Latin poetry in the style of Virgil after having only been trained on Virgil. Our best model generated 95.24% correct feet and 76.19% correct lines of dactylic hexameter and demonstrated ability to learn syntactic relationships—but was not fully semantically coherent across a full line or lines of text. This outperforms Lamar and Chambers' model in terms of quantitative metrics of meter but underperforms in terms of semantics. Our work highlights the limitations of low-resource models and small data sets for studying classical literature.

1 Key Information to include

- Mentor: Bessie Zhang
- Team Contributions: Both authors collaborated on all sections of the project. Jonathan led out on development of the data pipeline while August led out on development of the models.

2 Introduction

For millennia, Virgil's poems have been regarded as classic texts across a diverse set of cultures. As works of literature, they hold compelling insights into the human experience. Their cultural value has prompted rigorous study into how the texts function stylistically. Virgil's poems are famous for their disciplined and effective use of dactylic hexameter. Dactylic hexameter is a complex meter used in classic poetry. A valid line of dactylic hexameter has six feet, where each foot is a dactyl or a spondee. A dactyl is a long syllable followed by two short syllables, and a spondee is two long syllables. Defined as such, lines of dactylic hexameter can vary in word and syllable counts. This high level of variability along with the precise definition of correctness makes this meter more complex than other traditional poetic meters.

In 2019, Lamar and Chambers claimed that an encoder-decoder long short-term memory (LSTM) network could generate Greek poetry in the style of Homer with reasonably accurate meter but poor semantics after being trained exclusively on Homer (Lamar and Chambers, 2019). We attempted the more challenging, but similar, problem of training an encoder-decoder LSTM network to produce Latin poetry in the style of Virgil using only Virgil as training data. The added difficulty of our project comes from the fact that Virgil's corpus is approximately half the size of Homer's by line count. For

our training and testing data, we had access to fewer than thirteen thousand lines. The small data set along with the complicated task of generating Latin poetry in a difficult meter makes our problem an interesting litmus test for the power of encoder-decoder LSTM networks.

No other scholarly work has been published directly attempting our task.

To approach our task, we first tokenized Virgil's corpus: The Aeneid, The Georgics, and The Eclogues. We created embeddings for each token and then separated the training and testing data. Using the training data, we trained an encoder-decoder LSTM. With the testing data, we fed the trained model a line and had it generate output which we evaluated quantitatively for metric accuracy and qualitatively for syntax and semantics.

Our best model achieved 95.24% correctness in generated feet and 76.19% correctness in generated lines of dactylic hexameter. Lines were not coherent overall, but within phrases and for some context clues across lines, our model shows signs of the ability to make use of semantic and syntactic information. The rest of our models showed clear signs of overfitting and often repeated the same few lines. This clearly showed that these models could not condition its output on the input line.

This project highlighted the difficulties of developing a proficient low-resource model for tasks with limited amounts of task-specific data. However, we still demonstrated some success. Finding success with smaller, more interpretable models is important in order to productively use natural language processing tools in classical literature analysis.

3 Related Work

Modern poetry generation began with rule-based approaches but has since progressed almost exclusively to neural network approaches (Lamar and Chambers, 2019). Several papers have used similar modeling approaches to generate styles of poetry other than Virgil. Zhang and Lapata (2014) implemented character-level recurrent neural networks to generate Chinese poetry. More similarly to our approach, Wang et al. (2016) generated Chinese poetry using a bidirectional LSTM network. Lau et al used an LSTM to generate English poetry in iambic meter. All this work was done prior to 2019 when Lamar and Chambers published their paper on the generation of Homeric poetry which inspired our work. Lamar and Chambers trained a bidirectional encoder-decoder LSTM network on Homer's poetry. From thirty model generated lines, 91.3% of feet were properly formed and 43.33% of lines were correct dactylic hexameter.

Since then, large language models have far surpassed this level of performance. One can prompt ChatGPT 3.5 to generate poetry in the style of Virgil with impressive results. For our project, we were explicitly interested in whether an encoder-decoder LSTM network could accomplish this task having only been trained on Virgil. This was in part because we wanted to test whether Lamar and Chamber's approach would carry over to Latin and Virgil and also because we wanted to test the ability of a low-resource model trained only on task-specific data to accomplish this task. Furthermore, a smaller model leaves the door open for greater interpretability. Interpretability is particularly desirable for this task because a large portion of poetry in dactylic hexameter was performed live. Understanding how the model generates the correct meter may give insight into how humans created valid lines in live performances.

Our work serves as a rigorous attempt to validate Lamar and Chamber's results in a novel setting and an intriguing litmus test of the abilities of encoder-decoder LSTM networks.

4 Approach

We created and tested thirteen different models on this task. The code for each model was adapted from Assignment 4, but the vast majority of the code was modified or rewritten. Additional code was written to interface with our custom data processing pipeline. Each of our thirteen models are variations of our baseline model which was designed to be nearly identical to the model described by Lamar and Chambers. The thirteen models vary in the number of layers in the LSTM network, dropout rate, weight decay in the Adam optimizer, and hidden size of the encoder and decoder.

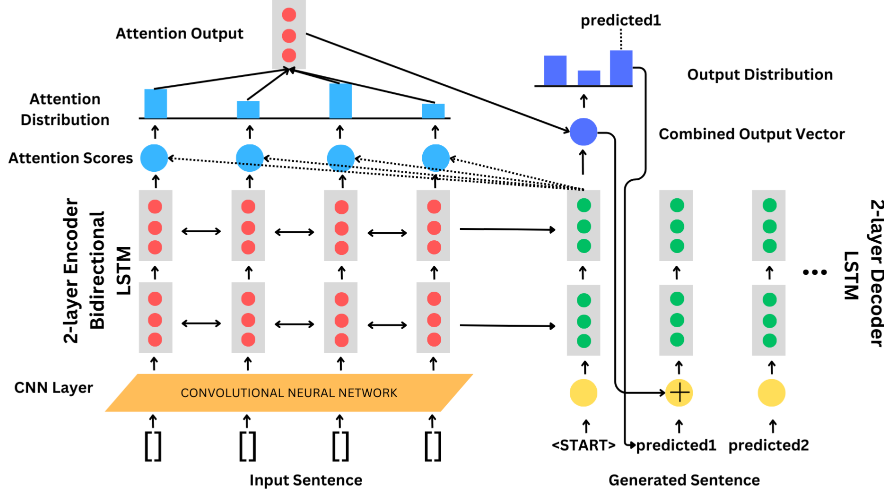

Figure 1: Baseline Model Architecture

As depicted above, our baseline model is a two-layer bidirectional encoder-decoder LSTM network with multiplicative attention. While not explicitly mentioned in the Lamar and Chambers paper, our model also includes a convolutional neural network layer between the embeddings and the encoder LSTM. Adding this gave improvement in early model attempts, so we kept this for all models. The hidden size is 100 for the encoder and 500 for the decoder.

Our model is described by equations adapted from Assignment 4.

The encoder hidden and cell states for each layer are defined as follows, where m is the maximum length of any input sentence:

$$\mathrm{h}_i^{enc1} = \left[\overrightarrow{\mathbf{h}_i^{enc1}}; \overleftarrow{\mathbf{h}_i^{enc1}}\right] \; where \; \mathbf{h}_i^{enc1} \in \mathbb{R}^{2h\times1}, \overrightarrow{\mathbf{h}_i^{enc1}}, \overleftarrow{\mathbf{h}_i^{enc1}} \in \mathbb{R}^{h\times1} \quad 1 \le i \le m$$
$$\mathbf{h}_i^{enc2} = \left[\overrightarrow{\mathbf{h}_i^{enc2}}; \overleftarrow{\mathbf{h}_i^{enc2}}\right] \; where \; \mathbf{h}_i^{enc2} \in \mathbb{R}^{2h\times1}, \overrightarrow{\mathbf{h}_i^{enc2}}, \overleftarrow{\mathbf{h}_i^{enc2}} \in \mathbb{R}^{h\times1} \quad 1 \le i \le m$$
$$\mathbf{c}_i^{enc1} = \left[\overrightarrow{\mathbf{c}_i^{enc1}}; \overleftarrow{\mathbf{c}_i^{enc1}}\right] \; where \; \mathbf{c}_i^{enc1} \in \mathbb{R}^{2h\times1}, \overrightarrow{\mathbf{c}_i^{enc1}}, \overleftarrow{\mathbf{c}_i^{enc1}} \in \mathbb{R}^{h\times1} \quad 1 \le i \le m$$
$$\mathbf{c}_i^{enc2} = \left[\overrightarrow{\mathbf{c}_i^{enc2}}; \overleftarrow{\mathbf{c}_i^{enc2}}\right] \; where \; \mathbf{c}_i^{enc2} \in \mathbb{R}^{2h\times1}, \overrightarrow{\mathbf{c}_i^{enc2}}, \overleftarrow{\mathbf{c}_i^{enc2}} \in \mathbb{R}^{h\times1} \quad 1 \le i \le m$$

The decoder hidden and cell states for each layer are defined as follows:

$$\mathrm{h}_0^{dec1} = \mathbf{W}_{h1}\left[\overrightarrow{\mathbf{h}_m^{enc1}}; \overleftarrow{\mathbf{h}_1^{enc1}}\right] \; where \; \mathbf{h}_0^{dec1} \in \mathbb{R}^{5h\times1}, \mathbf{W}_{h1} \in \mathbb{R}^{5h\times2h}$$
$$\mathbf{h}_0^{dec2} = \mathbf{W}_{h2}\left[\overrightarrow{\mathbf{h}_m^{enc2}}; \overleftarrow{\mathbf{h}_1^{enc2}}\right] \; where \; \mathbf{h}_0^{dec2} \in \mathbb{R}^{5h\times1}, \mathbf{W}_{h2} \in \mathbb{R}^{5h\times2h}$$
$$\mathbf{c}_0^{dec1} = \mathbf{W}_{c1}\left[\overrightarrow{\mathbf{c}_m^{enc1}}; \overleftarrow{\mathbf{c}_1^{enc1}}\right] \; where \; \mathbf{c}_0^{dec1} \in \mathbb{R}^{5h\times1}, \mathbf{W}_{c1} \in \mathbb{R}^{5h\times2h}$$
$$\mathbf{c}_0^{dec2} = \mathbf{W}_{c2}\left[\overrightarrow{\mathbf{c}_m^{enc2}}; \overleftarrow{\mathbf{c}_1^{enc2}}\right] \; where \; \mathbf{c}_0^{dec2} \in \mathbb{R}^{5h\times1}, \mathbf{W}_{c2} \in \mathbb{R}^{5h\times2h}$$

The decoder for each layer takes in $\overline{\mathbf{y}_t}$, a concatenation of the combined output vector $\mathbf{o}_{t-1}$ and the predicted word $\mathbf{y}_t$, as well as the previous hidden and cell states of the associated layer.

$$\mathrm{h}_t^{dec1}, \mathbf{c}_t^{dec1} = Decoder1\left(\overline{\mathbf{y}_t}, \mathbf{h}_{t-1}^{dec1}, \mathbf{c}_{t-1}^{dec1}\right) \; where \; \mathbf{h}_t^{dec1}, \mathbf{c}_t^{dec1}, \in \mathbb{R}^{5h\times1}$$
$$\mathrm{h}_t^{dec2}, \mathbf{c}_t^{dec2} = Decoder2\left(\overline{\mathbf{y}_t}, \mathbf{h}_{t-1}^{dec2}, \mathbf{c}_{t-1}^{dec2}\right) \; where \; \mathbf{h}_t^{dec2}, \mathbf{c}_t^{dec2} \in \mathbb{R}^{5h\times1}$$

Multiplicative attention is calculated as follows using the hidden state of the second layer. $\mathbf{e}_t$ is the attention score, α_t is the attention distribution, and $\mathbf{a}_t$ is the attention output.

$$\mathrm{e}_{t,i} = \left(\mathbf{h}_t^{dec2}\right)^T \mathbf{W}_{attProj}\mathbf{h}_i^{enc2} \; where \; \mathbf{e}_t \in \mathbb{R}^{m\times1}, \mathbf{W}_{attProj} \in \mathbb{R}^{5h\times2h} \quad 1 \le i \le m$$
$$\alpha_t = softmax\left(\mathbf{e}_t\right) \; where \; \alpha_t \in \mathbb{R}^{m\times1}$$

$\mathbf{a}_t = \sum_{i=1}^{m} \alpha_{t,i} \mathbf{h}_i^{enc2} \ where \ \mathbf{a}_t \in \mathbb{R}^{2h \times 1}$

The combined output vector is produced by the second layer of the decoder as follows.

$$\mathrm{u}_t = \left[\mathbf{h}_t^{\mathrm{dec2}}; \mathbf{a}_t\right] \ where \ \mathbf{u}_t \in \mathbb{R}^{7h \times 1}$$
$$\mathbf{v}_t = \mathbf{W}_u \mathbf{u}_t \ where \ \mathbf{v}_t \in \mathbb{R}^{5h \times 1}, \mathbf{W}_u \in \mathbb{R}^{5h \times 7h}$$
$$\mathbf{o}_t = dropout\left(\tanh\left(\mathbf{v}_t\right)\right) \ where \ \mathbf{o}_t \in \mathbb{R}^{5h \times 1}$$

The probability of predicting each token in the vocabulary V_t is computed as follows.

$$\mathrm{P}_t = softmax\left(\mathbf{W}_{vocab} \mathbf{o}_t\right) \ where \ \mathbf{P}_t \in \mathbb{R}^{V_t \times 1}, \mathbf{W}_{vocab} \in \mathbb{R}^{V_t \times 5h}$$

For loss, we calculated the cross entropy accordingly where $\mathbf{g}_t$ is a one-hot vector representing the predicted word.

$$\mathrm{J}_t(\theta) = CrossEntropy\left(\mathbf{P}_t, \mathbf{g}_t\right)$$

Models were implemented according to this architecture and these equations, where some models varied in number of layers and hidden sizes.

5 Experiments

We first created a pilot model to test our data pipeline and debug the basic model architecture. This pilot model differed from following models in that we used word-level tokenization and learned the embeddings using only Virgil's corpus. We added functionality to the word-level tokenizer in order to more correctly tokenize words with clitic particles into separate tokens instead of leaving them as a single token (Toolkit, 2024). All proceeding models used the following data pipeline from Figure 2.

5.1 Data Pipeline

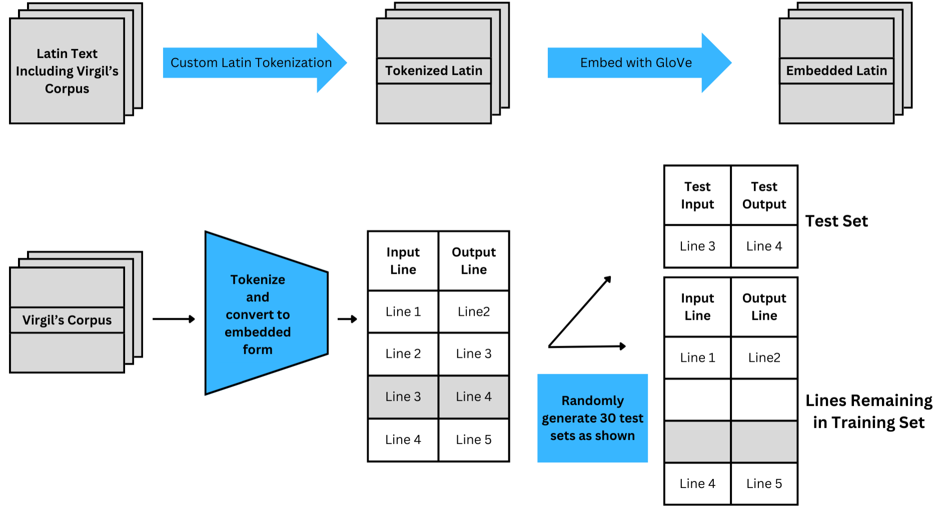

Figure 2: Data Pipeline

Download Latin Corpus: We downloaded a 51 MB corpus of Latin text from the Tesserae Project (2019). Included in this corpus are all of Virgil's works along with several hundred other works across various time periods.

Tokenize: We trained a byte-pair encoding (BPE) tokenizer using the HuggingFace Tokenizers library (Moi and Patry, 2023). The training dataset for the tokenizer was the entire Tesserae corpus

concatenated into a single text file.

Embed with Glove: After tokenizing the Tesserae corpus, we used GloVe to learn embeddings of size 50 for the whole corpus (Pennington et al., 2014).

Convert Virgil to Embedded Form: Using our trained tokenizer, we then tokenized only Virgil's corpus and used our learned embeddings to create an embedded version of Virgil's corpus.

Split into Training and Testing Data Sets: We iterated over the corpus to create input-output pairs. Each line in the corpus is an input line and the line following a given input line is its output line, together forming an input-output pair. When training, the model should learn to take in a line and accurately generate the line that follows it. Thirty of these pairs were randomly selected from the corpus and saved for the test set. For each test set pair, the pair remaining in the training data that contains the input line of the test pair as its output line was excluded from the training set such that the model never sees any part of the test input.

We then trained the model on the training data using teacher forcing. At inference time, the model was given an input line from the test set. The model generated an output line using beam search, never having seen the associated output line (i.e. no teacher forcing).

5.2 Evaluation method

To evaluate generated lines, we manually calculated the percentage of correct feet as well as the percentage of lines in perfect dactylic hexameter across the thirty generated lines produced by each model. For evaluation of the model's ability to generate semantically coherent lines, we manually compared the lines that the model generated with the input line and the correct line that it should have generated from the corpus to determine if the generated line's content made sense. The individual meanings of smaller phrases within the text and overall ability to produce syntactically correct Latin was also considered. These observations are qualitative in nature and included in the analysis, but do not have numerical measurements to compare with a baseline.

5.3 Experimental details

We wrote code for and trained thirteen models. All models were trained with an initial learning rate of 5e-4, much smaller than Lamar and Chamber's initial rate of 0.01. Without this adjustment, loss skyrocketed. Each model had either one, two, or three layers. Additionally, across models we varied the dropout rate of the model and the weight decay of the Adam optimizer. All models were trained until progress stalled according to the procedure of Assignment 4. The specifications for each of the thirteen models can be found in Appendix A.

5.4 Results

Many models showed clear signs of overfitting and simply repeated a small set of lines. These lines technically scored well according to our quantitative metrics, but they were clearly evidence of a model that couldn't condition its output well on input lines. Other models were designed to avoid overfitting (smaller hidden sizes, higher dropout rates, and weight decay), but these failed to learn the correct meter.

The only model that demonstrated the ability to generate new lines in response to input lines and performed well by our quantitative metrics was the three layer model with no weight decay, a dropout rate of 0.3, and hidden sizes of 100 for the encoder and 500 for the decoder. We compare the performance of this model to our pilot model and the performance of Lamar and Chamber's model below:

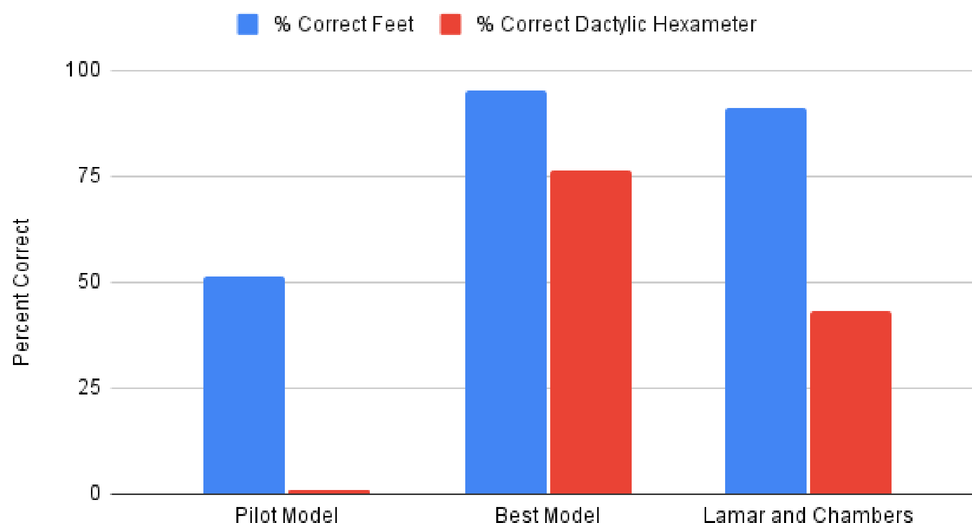

Figure 3: Relative Model Performance

Our pilot model had two layers, a dropout rate of 0.3, no weight decay, no convolutional layer, and word-level tokenization. The modifications we made from our pilot model to our top model led to significant improvements. Our top model also outperformed Lamar and Chambers in these quantitative metrics.

6 Analysis

The best model shows the ability to learn metrically convenient chunks of text, understand some semantic context, and adhere to syntactic rules. In the pair of examples in Figure 4, both input lines contain quotations of speech by a character in the poem. In the raw text, these lines would contain punctuation to make this clear, but during tokenization, we removed all punctuation. However, in both cases, the model predicted the phrase *Dixerat et Aenean*, "and he said, Aeneas. . . ," which shows both the model's ability to recognize that someone was speaking in the preceding line. In the first input line, the word inquit is used, which is often used for quotes, meaning "he said/says". This model may have learned this semantic cue.

In: *O miserae quas non manus* ***inquit*** *Achaca bello*

Out*: ***Dixerat et Aenean*** *Mnestheus acerque Serestus*

In: *Euryale infelix qua te regione reliqui*

Out*: ***Dixerat et Aenean*** *Phrygiam de gente sacerdos*

*Model generated.

Figure 4: Model recognizes semantic features

The next pair of examples in Figure 5 demonstrates the model's prioritization of creating metrical chunks. We see again that the model begins multiple lines with the same phrase *Dardanidae*. This a dactyl and a long syllable, allowing for either a dactyl or spondee in the second foot. This may be

seen as optimizing for metrical correctness as it gives the model more versatility with the generation of the next word. The last phrase in each of these lines are both metrically equivalent and quite typical of endings in dactylic hexameter but display differences in the model's use of grammar. *Sacerdos* must not only agree with the adjective *fortem* in grammatical gender, but it must also agree in grammatical case. Therefore, *sacerdos* should be *sacerdotem*, but this would not metrically fit into the foot. It is possible that the model is optimizing for meter over grammatical agreement in this case. The model does in fact demonstrate the capability for grammatical agreement in the second example. The preposition *de* requires a specific form of the adjectives (*gente*) and noun (*Latina*) that follow it, which is correctly generated by the model. Furthermore, *Latina* correctly agrees in grammatical gender with the noun it modifies, *gente*. Therefore, the model appears to have some understanding of semantics, syntax, and meter but strongly optimizes for meter. While not specifically compared in these examples, none of the model's generated sentences fit in with the larger context of the poem in the position they were intended to fill. While meter was nearly perfect and demonstrated local syntactic and semantic capabilities, the "big picture" details were missed.

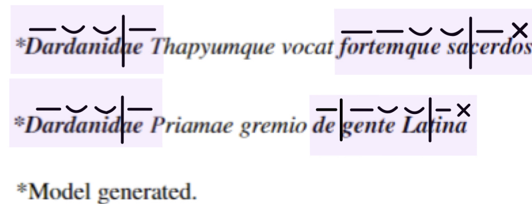

Figure 5: Model optimizes meter

7 Conclusion

Our best encoder-decoder LSTM network was able to consistently generate lines of Latin poetry in correct dactylic hexameter. The performance of this model in terms of meter outperformed both our pilot model and Lamar and Chambers' model. Overall, our model was successful in capturing the Virgil's meter and some syntactic information but struggled to generate semantically coherent lines. The primary limitation on our model's success was the small training dataset. However, the small dataset was integral to the task we set out to tackle in the first place. This experience suggests that large language models are likely necessary for some tasks for which there is little task-specific data. Use of large language models comes at the cost of interpretability, which places constraints on the usefulness of natural language processing approaches to studying classical literature.

References

Annie Lamar and America Chambers. 2019. Generating homeric poetry with deep neural networks. In *2019 First International Conference on Transdisciplinary AI (TransAI)*, pages 68–75.

Anthony Moi and Nicolas Patry. 2023. HuggingFace's Tokenizers.

Jeffrey Pennington, Richard Socher, and Christopher D. Manning. 2014. Glove: Global vectors for word representation. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543.

Classical Language Toolkit Tesserae Project. 2019. Cltk tesserae latin corpus.

The Classical Language Toolkit. 2024. The classical language toolkit.

Qixin Wang, Tianyi Luo, Dong Wang, and Chao Xing. 2016. Chinese song iambics generation with neural attention-based model.

Xingxing Zhang and Mirella Lapata. 2014. Chinese poetry generation with recurrent neural networks. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 670–680, Doha, Qatar. Association for Computational Linguistics.

A Appendix

A table of the specifications for each model trained is below. An “x” in a cell signifies that that hyperparameter value was used for training that instance of the model.

	Hidden Sizes (encoder/decoder		Dropout Rate		Weight Decay in Adam Optimizer	
	100/500	100/100	0.3	0.5	None	0.001
Single Layer Models	x		x		x	
	x		x			x
	x			x		x
		x		x		x
Two Layer Models	x		x		x	
	x		x			x
	x			x		x
		x		x		x
Pilot Model	x		x		x	
Three Layer Models	x		x		x	
	x		x			x
	x			x		x
		x		x		x

Model Training Parameters